Autonomous Agents and Multi-Agent Systems (2020) 34:47
https://doi.org/10.1007/s10458-020-09473-8

Specification testing of agent-based simulation using property-based testing

Jonathan Thaler[1,2] · **Peer-Olaf Siebers**[1]

Abstract

The importance of Agent-Based Simulation (ABS) as scientific method to generate data for scientific models in general and for informed policy decisions in particular has been widely recognised. However, the important technique of code testing of implementations like unit testing has not generated much research interested so far. As a possible solution, in previous work we have explored the conceptual use of *property-based testing*. In this code testing method, model specifications and invariants are expressed directly in code and tested through *automated* and *randomised* test data generation. This paper expands on our previous work and explores how to use property-based testing on a technical level to encode and test specifications of ABS. As use case the simple agent-based SIR model is used, where it is shown how to test agent behaviour, transition probabilities and model invariants. The outcome are specifications expressed directly in code, which relate whole classes of random input to expected classes of output. During test execution, random test data is generated automatically, potentially covering the equivalent of thousands of unit tests, run within seconds on modern hardware. This makes property-based testing in the context of ABS strictly more powerful than unit testing, as it is a much more natural fit due to its stochastic nature.

Keywords Agent-based simulation testing · Code testing · Test driven development · Model specification

1 Introduction

Since its inception in the early 1990s [17, 31, 36], Agent-Based Simulation (ABS) as a third way of doing science [3, 5] has matured substantially and has found its way into the mainstream of science [25]. Further, a number of ABS frameworks and tools like RePast, AnyLogic and NetLogo as well as open databases of ABS models [16] have been developed, allowing for quick and robust prototyping and development of models.

✉ Jonathan Thaler
jonathan.thaler@nottingham.ac.uk; jonathan.thaler@fhv.at

[1] School Of Computer Science, University of Nottingham, 7301 Wollaton Rd, Nottingham, UK

[2] Department of Computer Science, FH Vorarlberg, Hochschulstrasse 1, 6850 Dornbirn, Austria

Published online: 19 June 2020

However, despite the broad acceptance and adoption of ABS as methodology and *generative* way of doing science, there have been struggles as reported by Axelrod [4]. He discusses the vulnerability of ABS to misunderstanding: due to informal specifications of models and change requests amongst members of a research team, bugs are very likely to be introduced. Further, he reports how difficult it was to reproduce other work [2], which took the team four months, due to inconsistencies between the original code and the published paper. The consequence is that counter-intuitive simulation results can lead to weeks of checking whether the code matches the model and is bug-free [3].

The same problem was reported by researchers [7], which tried to reproduce the work of Gintis [19]. In his work, Gintis claimed to have found a mechanism in bilateral decentralized exchange, which resulted in Walrasian General Equilibrium without the neo-classical approach of a tatonement process through a central auctioneer [14]. This was a major breakthrough for economics as the theory of Walrasian General Equilibrium is non-constructive. It postulates the properties and existence of the equilibrium but does not explain the process and dynamics through which this equilibrium can be reached or constructed. Gintis seemed to have found a model for this process.

The authors [7] failed to reproduce the results and were only able to solve the problem by directly contacting Gintis, which provided the code, the definitive formal reference. It was found that there was a bug in the code leading to unexpected results, which were seriously damaged through this error. They also reported ambiguity between the informal model description in Gintis' paper and the actual implementation. This discovery lead to research in a functional framework for agent-based models of exchange [8], which tried to give a very formal functional specification of the model, coming very close to an implementation in Haskell. The failure of Gintis was investigated in more depth also by other researchers [18] who got access to Gintis' code through his website [20]. They found that the code in Object Pascal did not follow good object-oriented design principles (all of it was public, code duplication) and discovered a number of bugs serious enough to damage the results.

These issues show that due to the fact that ABS is primarily used for scientific research, often producing break-through scientific results, besides on converging both on standards for testing the robustness of implementations and on its tools, ABS more importantly needs to be *free of bugs*, *verified against their specification*, *validated against hypotheses* and ultimately be *reproducible* [4]. Further, a special issue with ABS is that the emergent behaviour of the system is generally not known in advance and researchers look for some *unique* emergent pattern in the dynamics. Whether the emergent pattern is then truly due to the system working correctly, or a bug in disguise is often not obvious and becomes increasingly difficult to assess with increasing system complexity.

These facts are also underlined in summaries of various ABS development methods [28] which all put fundamental emphasis on the verification and validation process for ABS. Although there exist methods and research of verification and validation in ABS, unfortunately, as Sect. 2 shows, there does not exist much research on the issue of code testing an ABS implementation. In software engineering, this task has been traditionally achieved by unit testing, as introduced by Beck in the seminal work on Test-Driven Development [6]. Unit tests are code pieces which test a given unit of functionality of some given feature. Generally, this results in hundreds or sometimes thousands of unit tests as all execution paths of the whole software should be covered.

We hypothesise that the reason why unit testing is not very present in the field of ABS verification and validation research, is a conceptual mismatch between unit testings' deterministic and ABS' rather stochastic nature. The fact that a unit test needs to be written for

each edge case makes it difficult to scale up to the stochastic nature of ABS, where the agent and model behaviour in general is often characterised by probabilistic distributions instead of deterministic rules. As a possible solution to this issue, our work [34] was the first to propose *property-based testing* as an alternative to unit testing for code testing ABS implementations. The main idea of property-based testing is to express model specifications and invariants directly in code and test them through *automated* and *randomised* test data generation. In our paper [34] we presented various ways to conceptually use property-based testing to code test ABS implementations. However, we did not discuss technical details and sequential statistical hypothesis testing and left the exact workings of property-based testing for ABS open as it was beyond the focus of that paper.

In this paper we pick up our conceptual work [34] and put it into a more technical perspective and demonstrate additional techniques of property-based testing in the context of ABS, which were not covered in the conceptual paper. More specifically, in this paper we additionally show how to encode agent specifications and model invariants into property tests, using an agent-based SIR model [24] as use case. Following an event-driven approach [27], we demonstrate how to express an agent specification in code by relating random input events to specific output events. Further, additionally using specific property-based testing features, which allow expressing expected coverage of data distributions, we show how transition probabilities can be tested. Finally, we also express model invariants by encoding them into property tests. By doing this, we demonstrate how property-based testing works on a technical level, how specifications and invariants can be put into code and how probabilities can be expressed and tested using statistically robust verification. This in-depth technical investigation was beyond the focus of our original, conceptual work [34] but the results of this paper gives additional evidence to its conclusion, that property-based testing maps naturally to ABS. Further, this work shows that in the context of ABS, property-based testing does scale up better than unit testing as it allows to run thousands of test cases automatically instead of constructing each manually and, more importantly, property-based testing is able to encode probabilities, something unit testing is not capable of in general.

The paper is structured as follows: Sect. 2 presents related work. In Sect. 3 property-based testing is introduced on a technical level. In Sect. 4 the agent-based SIR model is introduced, together with its informal event-driven specification. Sections 5 and 6 contain the main contribution of the paper, where it is shown how to encode agent specifications, transition probabilities and model invariants with property-based testing. Section 7 discusses the approach and concludes and Sect. 8 identifies further research.

2 Related work

Research on code testing of ABS is quite new with few publications so far. Our own work [34] is the first paper to introduce property-based testing to ABS. In it we show on a conceptual level that property-based testing allows to do both verification and validation of an implementation. However, we do not go into technical details of actual implementations, nor how to use property-based testing on a technical level, nor do we introduce the sequential statistical hypothesis testing of the QuickCheck library to express probabilities.

The use of unit testing in the context of ABS was first discussed by Collier et al. [15]. The authors introduce Test-Driven Development to ABS and use RePast to show how to

verify the correctness of an implementation with unit tests. A similar approach has been discussed for Discrete Event Simulation in the AnyLogic software toolkit [1].

Unit tests to verify an ABS implementation of maritime search operations was mentioned in [29]. The authors validate their model against an analytical solution from theory by running the simulation with unit tests and then performing a statistical comparison against the formal specification.

Property-based testing has also connections to data generators [21] and load generators and random testing [11] with the important benefit that property-based testing allows to express them directly in code.

The authors of [21] provide a case study of an agent-based simulation of synaptic connectivity, for demonstrating their generic testing framework in RePast and MASON, which rely on JUnit to run automated tests.

As most of these works are using unit testing, we provide a comparison between our proposed approach and unit testing in the following section.

3 Property-based testing

In property-based testing *functional specifications*, also called properties, are formulated in code and tried to falsify using a property-based testing library. In general, to falsify a functional specification, the property-based testing library runs *automated* test cases by *automatically* generating test data. When a test case fails, the functional specification was falsified by finding a counter example. For better analysis, the library then reduces the test data to its simplest form for which the test still fails, like shrinking of a list or pruning of a tree. On the other hand, if no counter example could be found for the functional specification, it is deemed valid and the test succeeds.

Property-based testing has its origins in the QuickCheck library [12, 13] of the pure functional programming language Haskell. QuickCheck tries to falsify the specifications by *randomly* sampling the test space. This library has been successfully used for testing Haskell code in the industry for years, underlining its maturity and real world relevance in general and of property-based testing in particular [22].

To give an understanding of how property-based testing works with QuickCheck, we give a practical example of how to implement a property of lists. Such a property is directly expressed as a function in Haskell, with the return type of `Bool`. This indicates whether the property holds for the given random inputs or not. In general, a QuickCheck property can take arbitrary inputs, with random data generated automatically by QuickCheck during testing. The example property we want to encode is that reversing a reversed list results again in the original list:

```
-- Reversing of a reversed list is the original list
prop_reverse_reverse :: [Int] -> Bool
prop_reverse_reverse xs = reverse (reverse xs) == xs
```

Testing the property with QuickCheck is simply done using the function `quickCheck`:

```
> quickCheck prop_reverse_reverse
+++ OK, passed 100 tests.
```

QuickCheck generates 100 test cases by default and requires all of them to pass. Indeed, all 100 test cases of `prop_reverse_reverse` pass and therefore the property as a whole passes the test. Note that we do not provide any data for the input argument `[Int]`, a list of Integers, because QuickCheck is doing this automatically for us. For the standard types of Haskell, QuickCheck provides existing data generators.

To give an example of what happens in case of failure due to a wrong property, we look at a wrong implementation of the property, that reverse distributes over the list append operator (++ in Haskell):

```
-- reverse is distributive over list append (++)
-- This is a wrong implementation for explanatory reasons!
-- For a correct property, swap xs and ys on the right hand side.
prop_reverse_distributive :: [Int] -> [Int] -> Bool
prop_reverse_distributive xs ys
  = reverse (xs ++ ys) == reverse xs ++ reverse ys

> quickCheck prop_reverse_distributive
*** Failed! Falsifiable (after 4 tests and 5 shrinks):
[0]
[1]
```

As expected, the property test fails because QuickCheck found a counter example to the property after 4 test cases. Also, we see that QuickCheck applied 5 shrinks to find the minimal failing counter example `xs = [0]` and `ys = [1]`. The reason for the failure is a wrong implementation of the `prop_reverse_distributive` property: to correct it, `xs` and `ys` need to be swapped on the right hand side of the equation. Note that when run repeatedly, QuickCheck might find the counter example earlier and might apply fewer shrinks due to a different random-number generator seed, resulting in different random data to start with.

3.1 Generators

QuickCheck comes with a lot of data generators for existing types like `String`, `Int`, `Double`, `[]` (`List`), but in case one wants to randomize custom data types, one has to write custom data generators. There are two ways to do this. The first one is to fix them at compile time by writing an `Arbitrary` type class instance. A type class can be understood as an interface definition, and an instance as a concrete implementation of such an interface for a specific type. The advantage of having an `Arbitrary` instance is that the custom data type can be used as random argument to a function as in the examples above. The second way to write custom data generators is to implement a run-time generator in the `Gen` context.

Here we implement a custom data generator for both cases, using a simple color representation as example. We start with the run-time option, running in the `Gen` context:

```
-- enumeration of colors
data Color = Red | Green | Blue

genColor :: Gen Color
genColor = elements [Red, Green, Blue]
```

This implementation makes use of the `elements :: [a] → Gen a` function, which picks a random element from a non-empty list with uniform probability. If a skewed distribution is needed, one can use the `frequency :: [(Int, Gen a)] → Gen a` function, where a frequency can be specified for each element. Generating on average 80% `Red`, 15% `Green` and 5% `Blue` can be achieved using this function:

```
genColor :: Gen Color
genColor = frequency [(80, Red), (15, Green), (5, Blue)]
```

Implementing an `Arbitrary` instance is straightforward, one only needs to implement the `arbitrary :: Gen a` method:

```
instance Arbitrary Color where
  arbitrary = genColor
```

When we have a random `Double` as input to a function, but want to restrict its random range to (0,1) because it reflects a probability, we can do this easily with `newtype` and implementing an `Arbitrary` instance:

```
newtype Probability = P Double

instance Arbitrary Probability where
  arbitrary = P <$> choose (0, 1)
```

3.2 Distributions

QuickCheck provides functions to measure the coverage of test cases. This can be done using the `label :: String → prop → Property` function. It takes a `String` as first argument and a testable property and constructs a `Property`. QuickCheck collects all the generated labels, counts their occurrences and reports their distribution. For example, it can be used to get an idea of the length of the random lists created in the `reverse_reverse` property shown above:

```
reverse_reverse_label :: [Int] -> Property
reverse_reverse_label xs
  = label ("length of random-list is " ++ show (length xs))
          (reverse (reverse xs) == xs)
```

When running the test, we get the following output:

```
+++ OK, passed 100 tests:
 5% length of random-list is 27
 5% length of random-list is 0
 4% length of random-list is 19
 ...
```

3.3 Coverage

QuickCheck provides two additional functions to work with test-case distributions: `cover` and `checkCoverage`. The function `cover :: Double → Bool → String → prop → Property` allows to explicitly specify that a given percentage of successful test cases belongs to a given class. The first argument is the expected percentage, the second argument is a `Bool` indicating whether the current test case belongs to the class or not, the third argument is a label for the coverage, and the fourth argument is the property which needs to hold for the test case to succeed.

Here we look at an example where we use `cover` to express that we expect 15% of all test cases to have a random list with at least 50 elements:

```
reverse_reverse_cover :: [Int] -> Property
reverse_reverse_cover xs
  = cover 15 (length xs >= 50) "Length of random list at least 50"
             (reverse (reverse xs) == xs)
```

When running the twice, we get the following output:

```
+++ OK, passed 100 tests (10% length of random list at least 50).
Only 10% Length of random-list at least 50, but expected 15%.
+++ OK, passed 100 tests (21% length of random list at least 50).
```

As can be seen, QuickCheck runs the default 100 test cases and prints a warning if the expected coverage is not reached. This is a useful feature, but it is up to us to decide whether 100 test cases are suitable and whether we can really claim that the given coverage will be reached or not. To free us from making this guess, QuickCheck provides the function `checkCoverage :: prop → Property`. When `checkCoverage` is used, QuickCheck will run an increasing number of test cases until it can decide whether the percentage in `cover` was reached or cannot be reached at all. The way QuickCheck does this, is by using sequential statistical hypothesis testing [35]. Thus, if QuickCheck comes to the conclusion that the given percentage can or cannot be reached, it is based on a robust statistical test giving us high confidence in the result.

When we run the example from above but now with `checkCoverage` we get the following output:

```
+++ OK, passed 12800 tests
    (15.445% length of random-list at least 50).
```

We see that after QuickCheck ran 12,800 tests it came to the statistically robust conclusion that, indeed, at least 15% of the test cases have a random list with at least 50 elements.

3.4 Comparison with unit testing

Section 2 shows that the standard in code testing of ABS is unit testing. For a better understanding and how our work relates to this other technique we briefly introduce unit testing in Java and compare it with property-based testing as introduced above.

As already pointed out in the introduction, unit tests are small pieces of code which test other code. These pieces of code are call test cases, and should be as small as possible, testing only a single aspect of the code under test. The way to implement unit tests is using the unit testing library JUnit, which provides annotations, assertions and test executors, to annotate test cases, express invariants, execute test cases and generate reports of the results.

In the following we briefly show how to express the properties of lists, as introduced above, with unit testing. We write a class `ListTest`, which contains all test cases, each annotated by `@Test`, which tells the test executor that this is a test to run. Invariants are expressed in our case with `assertEquals`, however JUnit provides all sorts of `asserts`, to express different invariants.

```
public class ListTest {
    // reverse of reverse restores the original order
    @Test
    public void testReverseReverse() {
        List<String> xs = new ArrayList<>();
        xs.add("Test1");
        xs.add("Test2");
        xs.add("Test3");

        // make a copy to compare original state
        List<String> xsOrig = new ArrayList<>(xs);

        // reverse twice, mutates xs
        Collections.reverse(xs);
        Collections.reverse(xs);

        // test invariant
        assertEquals(xs, xsOrig, "Lists not equal after reverse of reverse");
    }

    // reverse distributes over append (addAll in Java)
    @Test
    public void testReverseDistribute() {
        List<String> xs = new ArrayList<>();
        xs.add("A");
        xs.add("B");
        xs.add("C");

        List<String> ys = new ArrayList<>();
        ys.add("X");
        ys.add("Y");
        ys.add("Z");

        // copy lists
        List<String> xsCpy = new ArrayList<>(xs);
        List<String> ysCpy = new ArrayList<>(ys);

        // reverse (xs ++ ys) =>
        // append ys to xs, mutates xs

        xs.addAll(ys);
        // reverse xs, mutates xs
        Collections.reverse(xs);

        // reverse ys ++ reverse xs =>
        Collections.reverse(xsCpy);
        Collections.reverse(ysCpy);
        ysCpy.addAll(xsCpy);

        // express invariant (==)
        assertEquals(xs, ysCpy, "Lists not equal after reverse distributive");
    }
}
```

We immediately see how verbose unit tests are over property tests. The reason is not only found in object-oriented programming, but also that unit tests are not expressing specifications but following a very operational, imperative approach, stating *how* to test something instead of *what* is actually tested. We argue that without the comments added by us and appropriate naming of the tests, it would be not very obvious what exactly the unit tests are testing, whereas in property-based testing this is immediately clear.

A very important detail is that in this listing we only provide tests with 3 elements in each list. This does not cover all test cases, for example lists with a single element, empty lists, or lists of different sizes in the case of `testReverseDistribute` are missing. For a proper test coverage, we would need to manually provide all edge cases as additional test cases. This is implicitly covered in property-based testing, which generates the input data, automatically covering edge cases as well.

As for the `label`, `cover` and `checkCoverage` feature from property-based testing with QuickCheck, there is simply no equal in unit testing with JUnit. Therefore it is simply not possible to express such specifications.

It might look like that property-based testing is superior to unit testing, however it is not as both focus on different types of tests. Whereas property-based testing is ideally suited for testing data-centric problems, which can be expressed in specifications, such as the list properties above, unit testing is better suited for testing side effects of imperative code in a rather operational way. Therefore we see property-based testing and unit testing as complementary techniques.

4 Event-driven agent-based SIR model

As use case to develop the concepts in this paper, we use the explanatory SIR model [23]. It is a very well studied and understood compartment model from epidemiology, which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles spreading through a population.

In this model, people in a population of size N can be in either one of the three states *Susceptible*, *Infected* or *Recovered* at a particular time, where it is assumed that initially there is at least one infected person in the population. People interact *on average* with a given number of β other people per time unit and become infected with a given probability γ when interacting with an infected person. When infected, a person recovers *on average* after δ time units and is then immune to further infections. An interaction between infected persons does not lead to re-infection, thus, these interactions are ignored in this model. This definition gives rise to three compartments with the transitions seen in Fig. 1.

In this paper we follow [24] for translating the informal SIR specification into an event-driven agent-based approach [27]. The dynamics it produces are shown in Fig. 2, which was generated by our own implementation undertaken for this paper, accessible from our repository [32].

4.1 An informal specification

In this section we give an informal specification of the agent behaviour, relating the input to according output events. Before we can do that we first need to define the event types of the model, how they related to scheduling and how we can conceptually represent agents.

We are using Haskell as notation and implementation language as we conducted our research with it because it originated property-based testing. We are aware that Haskell is not a mainstream programming language, so to make this paper sufficiently self contained, we introduce concepts step-by-step, with many comments (– in the Haskell code) and explanations. This should allow readers, familiar with programming in general, understand the ideas behind what we are doing. Fortunately it is not necessary to go into detail of how agents are implemented as for our approach it is enough to understand the agents' inputs and outputs. For readers interested in the details of how to implement ABS in Haskell, we refer to another work of us [33].

We start by defining the states the agents can be in:

```
-- enumeration of the agents states
data SIRState = Susceptible | Infected | Recovered
```

The model uses three types of events. First, `MakeContact` is used by a susceptible agent to proactively make contact with β other agents per time unit by scheduling it to itself. Second, `Contact` is used by susceptible and infected agents to contact other agents, revealing their id and their state to the receiver. Third, `Recover` is used by an infected agent to proactively make the transition to recovered after δ time units.

```
-- agents are identified by a unique Integer
type AgentId = Int
-- enumeration of the three events
data SIREvent = MakeContact | Contact AgentId SIRState | Recover
```

As events are scheduled we need a new type to hold them, which we termed `QueueItem` as it is put into the event queue. It contains the event to be scheduled, the id of the receiving agent and the scheduling time.

```
type Time      = Double
data QueueItem = QueueItem SIREvent AgentId Time
```

Finally, we define an agent: it is a function, mapping an event to the current state of the agent with a list of scheduled events. This is a simplified view on how agents are actually implemented in Haskell but it suffices for our purpose.

Fig. 1 States and transitions in the SIR compartment model

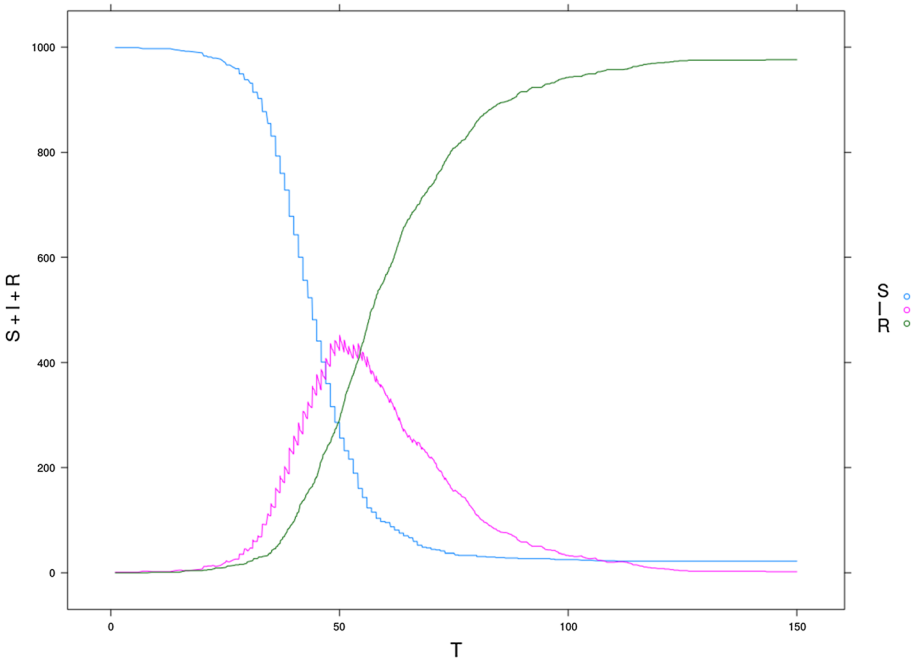

Fig. 2 Dynamics of the SIR compartment model using an event-driven agent-based approach. Population size $N = 1000$, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent

```
-- An agent maps an incoming event to the agents current state
-- and a list of scheduled events
sirAgent :: SIREvent -> (SIRState, [QueueItem])
```

We are now ready to give the full specification of the susceptible, infected and recovered agent by stating the input-to-output event relations. The susceptible agent is specified as follows:

1. `MakeContact`—if the agent receives this event it will output β `(Contact ai Susceptible)` events, where `ai` is the agents own id and `Susceptible` indicating the event comes from a susceptible agent. The events have to be scheduled immediately without delay, thus having the current time as scheduling timestamp. The receivers of the events are uniformly randomly chosen from the agent population. Additionally, to continue the pro-active contact making process, the agent schedules `MakeContact` to itself 1 time unit into the future. The agent doesn't change its state, stays `Susceptible` and does not schedule any other events than the ones mentioned.
2. `(Contact _ Infected)`—if the agent receives this event there is a chance of uniform probability γ that the agent becomes `Infected`. If this happens, the agent will schedule a `Recover` event to itself into the future, where the time is drawn randomly

from the exponential distribution with $\lambda = \delta$. If the agent does not become infected, it will not change its state, stays `Susceptible` and does not schedule any events.
3. `(Contact _ _)` or `Recover`—if the agent receives any of these other events it will not change its state, stays `Susceptible` and does not schedule any events.

This specification implicitly covers that a susceptible agent can never transition from a `Susceptible` to a `Recovered` state within a single event as it can only make the transition to `Infected` or stay `Susceptible`.

The infected agent is specified as follows:

1. `Recover`—if the agent receives this, it will not schedule any events but make the transition to the `Recovered` state.
2. `(Contact sender Susceptible)`—if the agent receives this, it will reply immediately with `(Contact ai Infected)` to `sender`, where `ai` is the infected agents' id and the scheduling timestamp is the current time. It will not schedule any events and stays `Infected`.
3. In case of any other event, the agent will not schedule any events and stays `Infected`.

This specification implicitly covers that an infected agent never goes back to the `Susceptible` state as it can only make the transition to `Recovered` or stay `Infected`. Also, from the specification of the susceptible agent it becomes clear that a susceptible agent who became infected, will always recover as the transition to `Infected` includes the scheduling of `Recovered` to itself.

The recovered agent specification is very simple: it stays `Recovered` forever and does not schedule any events.

5 Encoding agent specifications

We start by encoding the invariants of the susceptible agent directly into Haskell, implementing a function, which takes all necessary parameters and returns a `Bool` indicating whether the invariants hold or not. We are using pattern matching, therefore it reads like a formal specification due to the declarative nature of functional programming.

```
susceptibleProps :: SIREvent              -- random event sent to agent
                 -> SIRState              -- output state of the agent
                 -> [QueueItem SIREvent]  -- list of events the agent scheduled
                 -> AgentId               -- agent id of the agent
                 -> Bool
-- received Recover => stay Susceptible, no event scheduled
susceptibleProps Recover Susceptible es _ = null es
-- received Contact _ Recovered => stay Susceptible, no event scheduled
susceptibleProps (Contact _ Recovered) Susceptible es _ = null es
-- received Contact _ Susceptible => stay Susceptible, no event scheduled
susceptibleProps (Contact _ Susceptible) Susceptible es _  = null es
-- received Contact _ Infected, didn't get Infected, no event scheduled
susceptibleProps (Contact _ Infected) Susceptible es _ = null es
-- received Contact _ Infected AND got infected, check events
susceptibleProps (Contact _ Infected) Infected es ai
  = checkInfectedInvariants ai es
-- received MakeContact => stay Susceptible, check events
susceptibleProps MakeContact Susceptible es ai
  = checkMakeContactInvariants ai es cor
-- all other cases are invalid and result in a failed test case
susceptibleProps _ _ _ _ = False
```

Next, we give the implementation for the `checkInfectedInvariants` function. We omit a detailed implementation of `checkMakeContactInvariants` as it works in a similar way and its details do not add anything conceptually new. The function `checkInfectedInvariants` encodes the invariants which have to hold when the susceptible agent receives a `(Contact _ Infected)` event from an infected agent and becomes infected.

```
checkInfectedInvariants :: AgentId               -- agent id of the agent
                        -> [QueueItem SIREvent]  -- list of scheduled events
                        -> Bool
checkInfectedInvariants sender
  -- expect exactly one Recovery event
  [QueueItem receiver (Event Recover) t']
  -- receiver is sender (self) and scheduled into the future
  = sender == receiver && t' >= t
-- all other cases are invalid
checkInfectedInvariants _ _ = False
```

5.1 Writing a property test

After having encoded the invariants into a function, we need to write a QuickCheck property test, which calls this function with random test data. Although QuickCheck comes with a lot of data generators for existing Haskell types, it obviously does not have generators for custom types, like the `SIRState` and `SIREvent`. We refer to Sect. 3, where we explain the concept of data generators and implement generators for `Color` and `Probability`. The run-time generators for `SIRState` and `genEvent` for generating random `SIREvents` work similar to the `Color` generator and is omitted. For readers who are interested in a detailed implementation of both, we refer to the code repository [32].

All parameters to the property test are generated randomly, which expresses that the properties encoded in the previous section have to hold invariant of the model parameters. We make use of additional data generator modifiers: `Positive` ensures that a value

generated is positive; `NonEmptyList` ensures that a randomly generated list is not empty. Further, we use the function `label`, as explained in Sect. 3, to get an understanding of the distribution of the transitions. The case where the agents output state is `Recovered` is marked as "INVALID" as it must never occur, otherwise the test will fail, due to the invariants encoded in the previous section.

```
prop_susceptible :: Positive Int         -- beta (contact rate)
                 -> Probability          -- gamma (infectivity)
                 -> Positive Double      -- delta (illness duration)
                 -> Positive Double      -- current simulation time
                 -> NonEmptyList AgentId -- population agent ids
                 -> Gen Bool
prop_susceptible
  (Positive beta) (P gamma) (Positive delta) (Positive t) (NonEmpty ais) = do
  -- generate random event, requires the population agent ids
  evt <- genEvent ais
  -- run susceptible random agent with given parameters (implementation omitted)
  (ai, ao, es) <- genRunSusceptibleAgent beta gamma delta t ais evt
  -- check properties
  return (label (labelTestCase ao) (susceptibleProps evt ao es ai))
  where
     labelTestCase :: SIRState -> String
     labelTestCase Infected    = "Susceptible -> Infected"
     labelTestCase Susceptible = "Susceptible"
     labelTestCase Recovered   = "INVALID"
```

We have omitted the implementation of `genRunSusceptibleAgent` as it would require the discussion of implementation details of the agent. Conceptually speaking, it executes the agent with the respective arguments with a fresh random-number generator and returns the agent id, its state and scheduled events.

Finally, we run the test using QuickCheck. Due to the large random sampling space with 5 parameters, we increase the number of test cases to 100,000.

```
> quickCheckWith (stdArgs {maxSuccess=100000}) prop_susceptible
+++ OK, passed 100000 tests (6.77s):
94.522% Susceptible
 5.478% Susceptible -> Infected
```

All 100,000 test cases pass, taking 6.7 s to run on our hardware. The distribution of the transitions shows that we indeed cover both cases a susceptible agent can exhibit within one event. It either stays susceptible or makes the transition to infection. The fact that there is no transition to `Recovered` shows that the implementation is correct.

Encoding of the invariants and writing property tests for the infected agent follows the same idea and is not repeated here. Next, we show how to test transition probabilities using the powerful statistical hypothesis testing feature of QuickCheck.

5.2 Encoding transition probabilities

In the specifications from the previous section there are probabilistic state transitions, for example the susceptible agent *might* become infected, depending on the events it receives and the infectivity (γ) parameter. To encode these probabilistic properties we are using the function `cover` of QuickCheck. As introduced in Sect. 3, this function allows us to explicitly specify that a given percentage of successful test cases belong to a given class.

For our case we follow a slightly different approach than in the example of Sect. 3: we include all test cases into the expected coverage, setting the second parameter always to `True` as well as the last argument, as we are only interested in testing the coverage, which is in fact the property we want to test. Implementing this property test is then simply a matter of computing the probabilities and of case analysis over the random input event and the agents output. It is important to note that in this property test we cannot randomise the model parameters β, γ and δ because this would lead to random coverage. This might seem like a disadvantage but we do not really have a choice here, still the fixed model parameters can be adjusted arbitrarily and the property must still hold. We could have combined this test into the previous one but then we couldn't have used randomised model parameters. For this reason, and to keep the concerns separated, we opted for two different tests, which makes them also much more readable.

```
prop_susceptible_prob :: Positive Double       -- current simulation time
                      -> NonEmptyList AgentId  -- population agent ids
                                               -> Property
prop_susceptible_prob (Positive t) (NonEmpty ais) = do
  -- fixed model parameters, otherwise random coverage
  let cor = 5     -- contact rate (beta)
      inf = 0.05  -- infectivity (gamma)
      ild = 15.0  -- illness duration (delta)
  -- compute distributions for all cases depending on event and SIRState
  -- frequencies; technical detail, omitted for clarity reasons
  let recoverPerc       = ...
      makeContPerc      = ...
      contactRecPerc    = ...
      contactSusPerc    = ...
      contactInfSusPerc = ...
      contactInfInfPerc = ...
  -- generate a random event
  evt <- genEvent ais
  -- run susceptible random agent with given parameters, only
  -- interested in its output SIRState, ignore id and events
  (_, ao, _) <- genRunSusceptibleAgent cor inf ild t ais evt
  -- encode expected distributions
  -- case analysis over random input events
  return $ property $ case evt of
    Recover ->
      cover recoverPerc True "Susceptible recv Recover" True
    MakeContact ->
      cover makeContPerc True "Susceptible recv MakeContact" True
    (Contact _ Recovered) ->
      cover contactRecPerc True "Susceptible recv Contact * Recovered" True
    (Contact _ Susceptible) ->
      cover contactSusPerc True "Susceptible recv Contact * Susceptible" True
    (Contact _ Infected) ->
      -- case analysis over resulting agent state
      case ao of
        Susceptible ->
          cover contactInfSusPerc True
            "Susceptible recv Contact * Infected, stays Susceptible" True
        Infected ->
          cover contactInfInfPerc True
            "Susceptible recv Contact * Infected, becomes Infected" True
        _ ->
          cover 0 True "INVALID" True
```

We have omitted the details of computing the respective distributions of the cases, which depend on the frequencies of the events and the occurrences of `SIRState`

within the `Contact` event. By varying different distributions in the `genEvent` function, we can change the distribution of the test cases, leading to a more general test than just using uniform distributed events. When running the property test we get the following output:

```
+++ OK, passed 100 tests (0.01s):
40% Susceptible recv MakeContact
25% Susceptible recv Recover
14% Susceptible recv Contact * Infected, stays Susceptible
12% Susceptible recv Contact * Susceptible
 9% Susceptible recv Contact * Recovered

Only 9% Susceptible recv Contact * Recovered, but expected 11%
Only 25% Susceptible recv Recover, but expected 33%
```

QuickCheck runs 100 test cases, prints the distribution of the labels and issues warnings in the last two lines that generated and expected coverages differ in these cases. Further, not all cases are covered, for example the contact with an `Infected` agent and the case of becoming infected. The reason for these issues is insufficient testing coverage as 100 test cases are simply not enough for a statistically robust result. We could increase the number of test cases to 100,000, which *might* cover all cases but could still leave QuickCheck not satisfied as the expected and generated coverage *might* still differ.

As a solution to this fundamental problem, we use QuickChecks `checkCoverage` function. As introduced in Sect. 3, when the function `checkCoverage` is used, QuickCheck will run an increasing number of test cases until it can decide whether the percentage in `cover` was reached or cannot be reached at all. With the usage of `checkCoverage` we get the following output:

```
+++ OK, passed 819200 tests (7.32s):
33.3292% Susceptible recv Recover
33.2697% Susceptible recv MakeContact
11.1921% Susceptible recv Contact * Susceptible
11.1213% Susceptible recv Contact * Recovered
10.5356% Susceptible recv Contact * Infected, stays Susceptible
 0.5520% Susceptible recv Contact * Infected, becomes Infected
```

After 819,200 (!) test cases, run in 7.32 s on our hardware, QuickCheck comes to the statistically robust conclusion that the distributions generated by the test cases reflect the expected distributions and passes the property test.

6 Encoding model invariants

By informally reasoning about the agent specification and by realising that they are, in fact, a state machine with a one-directional flow of *Susceptible* → *Infected* → *Recovered* (as seen in Fig. 1), we can come up with a few invariants, which have to hold for any SIR simulation run, *under random model parameters* and independent of the random-number stream and the population:

1. Simulation time is monotonic increasing. Each event carries a timestamp when it is scheduled. This timestamp may stay constant between multiple events but will eventually increase and must never decrease. Obviously, this invariant is a fundamental assumption in most simulations where time advances into the future and does not flow backwards.
2. The number of total agents N stays constant. In the SIR model no dynamic creation or removal of agents during simulation happens.
3. The number of susceptible agents S is monotonic decreasing. Susceptible agents *might* become infected, reducing the total number of susceptible agents but they can never increase because neither an infected nor recovered agent can go back to susceptible.
4. The number of recovered agents R is monotonic increasing. This is because infected agents *will* recover, leading to an increase of recovered agents but once the recovered state is reached, there is no escape from it.
5. The number of infected agents I respects the invariant of the equation $I = N - (S + R)$ for every step. This follows directly from the first property which says $N = S + I + R$.

6.1 Encoding the invariants

All these properties are expressed directly in code and read like a formal specification due to the declarative nature of functional programming:

```
sirInvariants :: Int                     -- N total number of agents
              -> [(Time,(Int,Int,Int))] -- output each step: (Time,(S,I,R))
              -> Bool
sirInvariants n aos = timeInc && aConst && susDec && recInc && infInv
  where
    (ts, sirs)  = unzip aos    -- split Time and (S,I,R) into 2 separate lists
    (ss, _, rs) = unzip3 sirs  -- split S, I and R into 3 separate lists

    -- 1. time is monotonic increasing
    timeInc = allPairs (<=) ts
    -- 2. number of agents N stays constant in each step
    aConst = all agentCountInv sirs
    -- 3. number of susceptible S is monotonic decreasing
    susDec = allPairs (>=) ss
    -- 4. number of recovered R is monotonic increasing
    recInc = allPairs (<=) rs
    -- 5. number of infected I = N - (S + R)
    infInv = all infectedInv sirs

    -- encodes property 2
    agentCountInv :: (Int,Int,Int) -> Bool
    agentCountInv (s,i,r) = s + i + r == n

    -- encodes property 5
    infectedInv :: (Int,Int,Int) -> Bool
    infectedInv (s,i,r) = i == n - (s + r)

    -- returns True if a predicate p is satisfied for all pairs in a list
    allPairs :: (Ord a, Num a) => (a -> a -> Bool) -> [a] -> Bool
    allPairs p xs = all (uncurry f) (pairs xs)

    -- pair up neighbouring elements of a list
    pairs :: [a] -> [(a,a)]
    pairs xs = zip xs (tail xs)
```

Putting this property into a QuickCheck test is straightforward. We randomise the model parameters β (contact rate), γ (infectivity) and δ (illness duration) because the properties have to hold for all positive, finite model parameters.

```
prop_sir_invariants :: Positive Int     -- beta (contact rate)
                    -> Probability      -- gamma (infectivity)
                    -> Positive Double  -- delta (illness duration)
                    -> TimeRange        -- random duration in range (0, 50)
                    -> [SIRState]       -- population
                    -> Property
prop_sir_invariants
    (Positive beta) (P gamma) (Positive delta) (T t) as  = property (do
  -- total agent count
  let n = length as
  -- run the SIR simulation with a new RNG
  ret <- genSimulationSIR as beta gamma delta t
  -- check invariants and return result
  return (sirInvariants n ret)
```

Due to the large sampling space, we increase the number of test cases to run to 100,000 and all tests pass as expected. It is important to note that we put a random time limit within the range of (0,50) on the simulations to run. Meaning, that if a simulation does not terminate before that limit, it will be terminated at that random `t`. The reason for this is entirely practical as it ensures that the wall clock time to run the tests stays within reasonable bounds while still retaining randomness.

7 Discussion

In this paper we have shown how to use property-based testing on a technical level to encode informal specifications of agent behaviour and model invariants into formal specification directly in code. By incorporating this powerful technique into simulation development, confidence in the correctness of an implementation is likely to increase substantially, something of fundamental importance for ABS in general and for models supporting far-reaching policy decision in particular. Although our research uses the simple agent-based SIR model to demonstrate our approach, we hypothesise that it is applicable to event-driven ABS [27] in general, as we clearly focus on relating input to output events. To put our hypothesis to a test would require the generalisation of this simple model into a full framework of property-based testing for event-driven ABS, which we leave for further research.

The benefits of a property-based approach in ABS over unit testing is manifold. First, it expresses specifications rather than individual test cases, which makes it more general than unit testing. It allows expressing probabilities of various types (hypotheses, transitions, outputs) and performing statistically robust testing by sequential hypothesis testing. Most importantly, it relates whole classes of inputs to whole classes of outputs, automatically generating thousands of tests if necessary, therefore better scaling to the stochastic nature of ABS.

The main challenge of property-based testing is to write custom data generators, which produce sufficient coverage for the problem at hand, something not always obvious when starting out. Further, it is not always clear without some analysis, whether a property test actually covers enough of the random test space or not. As a robust solution to this issues, QuickCheck provides functions allowing to specify required coverage as well as

functionality to perform sequential statistical hypothesis testing to arrive at statistically robust coverage tests. An alternative solution to the potential coverage problems of QuickCheck is the *deterministic* property-testing library SmallCheck [30], which instead of randomly sampling the test space, enumerates test cases exhaustively up to some depth.

We hypothesise that it is very likely that if Gintis [19] would have applied rigorous unit and property-based testing to his model he might have found the inconsistencies and could have corrected them. Additionally, the code of the re-implementation [18] contains numerous invariant checks and assertions, which are properties expressed in code, thus immediately applicable for property-based testing. Further, due to the mathematical nature of Gintis' model, many properties in the form of formulas can be found in the paper specification [19], which could be directly expressible using property-based and unit testing.

Property-based testing has a close connection to model checking [26], where properties of a system are proved in a formal way. The important difference is that the checking happens directly on code and not on the abstract, formal model, thus one can say that it combines model checking and unit testing, embedding it directly in the software development and Test-Driven Development process without an intermediary step. We hypothesise that adding it to the already existing testing methods in the field of ABS is of substantial value as it allows to cover a much wider range of test cases due to automatic data generation. This can be used in two ways: to verify an implementation against a formal specification and to test hypotheses about an implemented simulation. This puts property-based testing on the same level as agent- and system testing, where not technical implementation details of agents are checked like in unit tests but their individual complete behaviour and the system behaviour as a whole.

8 Further research

The transitions we implemented were one-step transitions, feeding only a single event to the agents. Although we covered the full functionality by also testing the infected and recovered agent separately, the next step is to implement property tests which test the full transition from susceptible to recovered. This would require a stateful approach with multiple events and a different approach calculating the probabilities. We leave this for further research.

We have omitted tests for the infected agent as they follow conceptually the same patterns as the susceptible agent. The testing of transitions of the infected agent work slightly different though as they follow an exponential distribution but are encoded in a similar fashion as demonstrated with the susceptible agent. The case for the recovered agent is a bit more subtle, due to its behaviour: it simply stays `Recovered` *forever*. A property-based test for the recovered agent would therefore run a recovered agent for a random number of time units and require that its output is always `Recovered`. Of course, this is no proof that the recovered agent stays recovered *forever* as this would take forever to test and is thus not computable. Here we are hitting the limits of what is possible with random black-box testing like property-based testing. Without looking at the actual implementation it is not possible to prove that the recovered agent is really behaving as specified. We made this fact clear at the beginning of this paper, that property-based testing is not proof for correctness, but is only a support for raising the confidence in correctness by constructing cases that show that the behaviour is not incorrect. To be really sure that the recovered

agent behaves as specified we need to employ white-box verification and look at the actual implementation. This is beyond the scope of this paper and left for further research.

The reason why we limit the virtual time in Sect. 6 to 50 time units is also related to the limitations of property-based testing. Theoretically, limiting the duration is actually not necessary because we can reason that the SIR simulation *will always* reach an equilibrium in finite steps. Unfortunately, this is not possible to express and test directly with property-based testing and would also require a dependently typed programming language like Idris [9, 10]. We leave this for further research.

An interesting and valuable undertaking would be to conduct a user study with a couple of users (around 5) to show that our approach indeed brings benefits, for example injecting faults into implementations and then see if and how the users detect these faults using property-based testing. As a user study is beyond the focus of this paper, we leave it for further research.

References

1. Asta, S., Özcan, E., & Siebers, P. O. (2014). An investigation on test driven discrete event simulation. In *Operational research society simulation workshop 2014 (SW14)*. http://eprints.nottingham.ac.uk/28211/.
2. Axelrod, R. (1995). The convergence and stability of cultures: Local convergence and global polarization. Working paper, Santa Fe Institute. http://econpapers.repec.org/paper/wopsafiwp/95-03-028.htm.
3. Axelrod, R. (1997). Advancing the art of simulation in the social sciences. In R. Conte, R. Hegselmann, & P. Terna (Eds.), *Simulating social phenomena* (pp. 21–40). Berlin: Springer. https://doi.org/10.1007/978-3-662-03366-1_2.
4. Axelrod, R. (2006). Chapter 33 agent-based modeling as a bridge between disciplines. In L. T. A. K. L. Judd (Ed.), *Handbook of computational economics* (Vol. 2, pp. 1565–1584). Amsterdam: Elsevier. https://doi.org/10.1016/S1574-0021(05)02033-2.
5. Axelrod, R., & Tesfatsion, L. (2006). A guide for newcomers to agent-based modeling in the social sciences. Staff general research papers archive, Iowa State University, Department of Economics. http://econpapers.repec.org/paper/isugenres/12515.htm.
6. Beck, K. (2002). *Test Driven Development: By Example* (01st ed.). Boston: Addison-Wesley Professional.
7. Botta, N., Mandel, A., Hofmann, M., Schupp, S., & Ionescu, C. (2013). Mathematical specification of an agent-based model of exchange. In *Proceedings of the AISB convention*.
8. Botta, N., Mandel, A., Ionescu, C., Hofmann, M., Lincke, D., Schupp, S., et al. (2011). A functional framework for agent-based models of exchange. *Applied Mathematics and Computation*, *218*(8), 4025–4040. https://doi.org/10.1016/j.amc.2011.08.051.
9. Brady, E. (2013). Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, *23*(05), 552–593. https://doi.org/10.1017/S095679681300018X.
10. Brady, E. (2017). *Type-driven development with Idris*. New York: Manning Publications Company. **(Google-Books-ID: eWzEjwEACAAJ)** .
11. Burnstein, I. (2010). *Practical software testing: A process-oriented approach* (1st ed.). Berlin: Springer Publishing Company, Incorporated.
12. Claessen, K., & Hughes, J. (2000). QuickCheck—A lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on functional*

programming, ICFP '00 (pp. 268–279). New York, NY, USA: ACM. https://doi.org/10.1145/351240.351266.
13. Claessen, K., & Hughes, J. (2002). Testing monadic code with QuickCheck. *SIGPLAN Notices*, *37*(12), 47–59. https://doi.org/10.1145/636517.636527.
14. Colell, A. M. (1995). *Microeconomic theory*. Oxford: Oxford University Press. **(Google-Books-ID: dFS2AQAACAAJ)** .
15. Collier, N., & Ozik, J. (2013). Test-driven agent-based simulation development. In *2013 winter simulations conference (WSC)* (pp. 1551–1559). https://doi.org/10.1109/WSC.2013.6721538.
16. ComSES: Computational Model Library. (2019). Retrieved June 18, 2020 from https://www.comses.net/codebases/.
17. Epstein, J. M., & Axtell, R. (1996). *Growing artificial societies: Social science from the bottom up*. Washington, DC: The Brookings Institution.
18. Evensen, P., & Märdin, M. (2010). An extensible and scalable agent-based simulation of Barter economics. Master's thesis. Chalmers University of Technology, Göteborg. https://gupea.ub.gu.se/handle/2077/22063.
19. Gintis, H. (2006). The emergence of a price system from decentralized bilateral exchange. *Contributions in Theoretical Economics*, *6*(1), 1–15. https://doi.org/10.2202/1534-5971.1302.
20. Gintis, H. (2019). Herbert Gintis Website. Retrieved January 14, 2020 from https://people.umass.edu/gintis/. https://people.umass.edu/gintis/.
21. Gurcan, O., Dikenelli, O., & Bernon, C. (2013). A generic testing framework for agent-based simulation models. *Journal of Simulation*, *7*(3), 183–201. https://doi.org/10.1057/jos.2012.26.
22. Hughes, J. (2007). QuickCheck testing for fun and profit. In *Proceedings of the 9th international conference on practical aspects of declarative languages, PADL'07* (pp. 1–32). Berlin, Heidelberg: Springer-Verlag. https://doi.org/10.1007/978-3-540-69611-7_1. http://dx.doi.org/10.1007/978-3-540-69611-7_1.
23. Kermack, W. O., & McKendrick, A. G. (1927). A contribution to the mathematical theory of epidemics. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, *115*(772), 700–721. https://doi.org/10.1098/rspa.1927.0118.
24. Macal, C. M. (2010). To agent-based simulation from system dynamics. In *Proceedings of the winter simulation conference, WSC '10* (pp. 371–382). Baltimore, Maryland: Winter Simulation Conference. http://dl.acm.org/citation.cfm?id=2433508.2433551.
25. Macal, C. M. (2016). Everything you need to know about agent-based modelling and simulation. *Journal of Simulation*, *10*(2), 144–156. https://doi.org/10.1057/jos.2016.7.
26. McMillan, K. L. (1992). Symbolic model checking: An approach to the state explosion problem. Ph.D. thesis, USA . UMI Order No. GAX92-24209.
27. Meyer, R. (2014). Event-driven multi-agent simulation. In *Multi-agent-based simulation XV, lecture notes in computer science* (pp. 3–16). Cham: Springer. https://doi.org/10.1007/978-3-319-14627-0_1. https://link.springer.com/chapter/10.1007/978-3-319-14627-0_1.
28. North, M. J. (2018). Hammer or tongs: How best to build agent-based models? In Y. Demazeau, B. An, J. Bajo, & A. Fernández-Caballero (Eds.), *Advances in practical applications of agents, multi-agent systems, and complexity: The PAAMS collection* (pp. 3–11). Cham: Springer.
29. Onggo, B. S. S., & Karatas, M. (2016). Test-driven simulation modelling: A case study using agent-based maritime search-operation simulation. *European Journal of Operational Research*, *254*, 517–531. https://doi.org/10.1016/j.ejor.2016.03.050.
30. Runciman, C., Naylor, M., Lindblad, F. (2008). Smallcheck and lazy Smallcheck: Automatic exhaustive testing for small values. In *Proceedings of the first ACM SIGPLAN symposium on Haskell, Haskell '08* (pp. 37–48). New York, NY, USA: ACM. https://doi.org/10.1145/1411286.1411292.
31. Siebers, P.O., & Aickelin, U. (2008). Introduction to multi-agent simulation. arXiv:0803.3905 [cs].
32. Thaler, J. (2019). Repository of agent-based SIR implementation in Haskell. Retrieved June 18, 2020 fromhttps://github.com/thalerjonathan/haskell-sir.
33. Thaler, J., Altenkirch, T., & Siebers, P. O. (2018). Pure functional epidemics: An agent-based approach. In *Proceedings of the 30th symposium on implementation and application of functional languages, IFL 2018* (pp. 1–12). New York, NY, USA: ACM. https://doi.org/10.1145/3310232.3310372. http://doi.acm.org/10.1145/3310232.3310372. Event-place: Lowell, MA, USA.
34. Thaler, J., & Siebers, P. O. (2019). Show me your properties: The potential of property-based testing in agent-based simulation. In *Proceedings of the 2019 summer simulation conference, SummerSim '19* (pp. 1:1–1:12). San Diego, CA, USA: Society for Computer Simulation International. http://dl.acm.org/citation.cfm?id=3374138.3374139.

35. Wald, A. (1992). Sequential tests of statistical hypotheses. In S. Kotz & N. L. Johnson (Eds.), *Breakthroughs in statistics: Foundations and basic theory* (pp. 256–298)., Springer series in statistics New York, NY: Springer. https://doi.org/10.1007/978-1-4612-0919-5_18.
36. Wooldridge, M. (2009). *An introduction to multiagent systems* (2nd ed.). New York: Wiley.